LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-418135

Analyzing the Performance of Scientific Applications with Open|SpeedShop

M. Schulz, J. Galarowicz, D. Maghrak, W. Hachfeld, D. Montoya, S. Cranford

October 16, 2009

ParCFD
Moffet Field, CA, United States
May 18, 2009 through May 22, 2009

Disclaimer

COVER SHEET

NOTE: This conversheet is intended for you to list your article title and author(s) name only—this page will not appear on the CD-ROM.

Title: Analyzing the Performance of Scientific Applications with Open|SpeedShop

Authors: Martin Schulz[*], Jim Galarowicz[**], Don Maghrak[**], William Hachfeld[**], David Montoya[***], Scott Cranford[****]

PAPER DEADLINE: ***SEPTEMBER 30, 2009****

PAPER LENGTH: **16 PAGES (Maximum)**

ABSTRACT

Open|SpeedShop (O|SS) is a comprehensive, open source, parallel performance analysis tool set that provides the most common performance analysis steps in a single environment. It can help application programmers to analyze, understand and optimize the performance of their codes. O|SS provides a wide range of performance experiments covering both sampling and tracing techniques. It works on sequential, MPI, and multithreaded codes without requiring source code modifications. The latter allows users an easy integration of O|SS into their application's workflow and runtime environment. This ensures a low learning curve and is essential for achieving a wide spread adoption of any performance tool set. O|SS, together with documentation and extensive tutorials, is available at http://www.openspeedshop.org/.

INTRODUCTION

Performance analysis and optimization is a critical step in the development process of any scientific application. Efficient and easy-to-use tool support is essential to allow users to complete this task, yet current tools are often targeted for the performance analysis expert, and therefore cumbersome to use or require changes to the compilation or execution process. This makes these tools unattractive for application developers who often have limited time allocated for performance optimizations.

To overcome these problems, we have designed Open|SpeedShop (O|SS), an open source multi platform Linux performance tool that provides easy access to an application's performance profile, while not precluding more sophisticated and detailed analysis found in other tools. For this purpose it combines guided performance analysis through graphical wizards and preconfigured detailed analysis panels with low-level access to performance data through scripting interfaces and Python integration.

O|SS's performance analysis functionality includes a set of specific *Experiments* that allow the user to easily gather a variety of different performance statistics about

* Lawrence Livermore National Laboratory, Livermore, CA, USA
** Krell Institute, Ames, IA, USA
*** Los Alamos National Laboratory, Los Alamos, NM, USA
**** Sandia National Laboratories, Livermore, CA, USA

Emails: schulzm@llnl.gov, {jeg, dpm, wdh}@krellinst.org, dmont@lanl.gov, scranfo@sandia.gov

an application. This includes Program Counter (PC) sampling, a light weight way to get an overview of application performance bottlenecks; Call Stack Sampling analysis, a technique to find hot call paths; Hardware Performance Counters, providing access to low level information such as cache or TLB misses; MPI Profiling and Tracing, enabling users to detect MPI communication bottlenecks; I/O Profiling and Tracing to study an application's I/O characteristics; and Floating Point Exception (FPE) analysis to detect floating point exceptions that can slow down applications.

The tool set currently targets Linux clusters built using Intel or AMD processors and works on a wide variety of Linux distributions. It offers two methods for instrumentation and data collection: an offline option that instruments the application at job start, produces unprocessed raw files at runtime, and then automatically postprocesses these files for later analysis; and an online option that dynamically instruments the application at runtime and collects the data using a hierarchical overlay network. While the first option typically is easier to install and allows for less instrumentation overhead, the second option provides advanced scalability as well as new functionality like being able to attach to already running applications. In both cases, though, the data is stored in the form of a single relational database.

Once collected, O|SS displays the data through a set of detailed reports that allow the user to easily relate the performance information back to their application source code. This information is accessible through a comprehensive GUI, from a command line interface, as well as from within Python scripts. Additionally, the tool set includes a series of analysis techniques, including outlier detection, load balance analysis, and cross experiment comparisons. In summary, O|SS's functionality provides a comprehensive set of techniques that greatly aid in the analysis and understanding of parallel application performance.

The remainder of this paper is structured as follows: Section 3 describes the central concept for the use of O|SS, an *Experiment*. Section 4 illustrates O|SS's user interfaces and workflow, followed by a description of its architecture in Section 5 and a discussion of the two instrumentation options in Section 6. Section 7 shows the overheads caused by O|SS and Section 8 provides a quick overview of related work. We then conclude with a brief look into O|SS's future and some final remarks in Section 9.

THE CONCEPT OF EXPERIMENTS

The central concept in O|SS's workflow is an *Experiment*. It defines what is being measured and/or analyzed. Users select their experiment at the beginning of any performance analysis run depending on what kind of performance bottleneck they would like to investigate. With that, they implicitly choose how the data will be extracted from the application, in which form it will be stored, and what kind of views are available for its analysis.

By default, the tool set offers three sampling and three tracing experiments covering most common needs for performance analysis. These are

- **PC Sampling**, which provides a statistical overview of where a code spends its time and hence offers a good first overview of a code's behavior;
- **User Time**, which adds stack trace information to the gathered samples and thereby adds context on how a code reached the observed bottlenecks;

- **HardWare Counter (HWC)**, which enables users to measure information offered by the CPU's performance counters like cache or TLB misses;
- **I/O tracing**, which gathers a detailed trace of all POSIX I/O operations found in a code and the time they took;
- **MPI tracing**, which provides the same type of information, but for MPI calls executed during an applications execution; and
- **FPE tracing**, which tracks floating point exceptions triggered by the FPU.

Most of these experiments exist in several variants that allow the user to further define which/how much information is being collected. For example, the hardware counter experiment can optionally collect stack traces for each sample, which enables the user to distinguish different invocation contexts. Further, both the I/O and the MPI tracing experiments have the ability to include function arguments (such as number of bytes written during an I/O command or source rank for an MPI receive operation) in the traces. Finally, the MPI tracing experiment also enables the storage of MPI traces in the Open Trace Format (OTF) [9] for additional analysis in timeline visualization tools like Vampir [14].

Except for the MPI tracing experiments, which naturally requires parallel execution in an MPI environment, all other experiment can be used in either sequential or parallel (MPI or multithreaded) environments. In the latter case, O|SS keeps track of which performance information was generated by which process or thread and then offers the data either in an aggregated (global) view or on a per task/thread or task/thread group basis.

In addition to the existing default experiments, O|SS allows the addition of new experiments through a plugin infrastructure. By specifying separate plugins for data collection, data preparation, and data presentation programmers can add new tool functionality without having to reimplement the (often complex) surrounding infrastructure. This includes components like instrumentation, data conversion, transport, and storage, and user interfaces. This capability allows O|SS to be extensible and also meet specialized needs, e.g., for particular hardware platforms with special needs or performance monitoring capabilities.

USER INTERFACES AND WORKFLOW

O|SS offers three different user interface, which are shown in Figure 1: a Graphical User Interface (GUI) with source code browser and simple graphing and visualization panels; a specialized scripting language similar to ones used by debuggers like *gdb*, called *Command Line Interface* (CLI); and a Python module (pyO|SS) that allows the invocation of O|SS from Python scripts. Additionally, O|SS's default installation provides a set of convenience scripts that provide single command access to O|SS experiments without having to learn any command syntax or scripting languages.

It is important to note that all user interfaces are equivalent in their functionality and capabilities and can interact. Data gathered using one interface can be read and analyzed by any other. This full interoperability is made possible through O|SS's user interface architecture, which is shown in the top part of the graphics in Figure 2. All user interfaces are layered on top of a single user interface access component, which

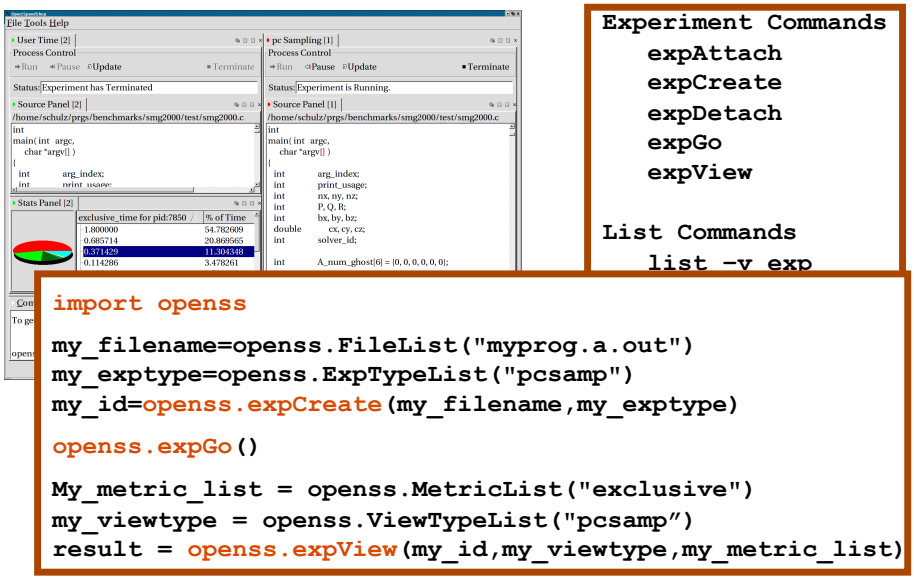

Figure 1. Open|SpeedShop provides three interoperable user interfaces.

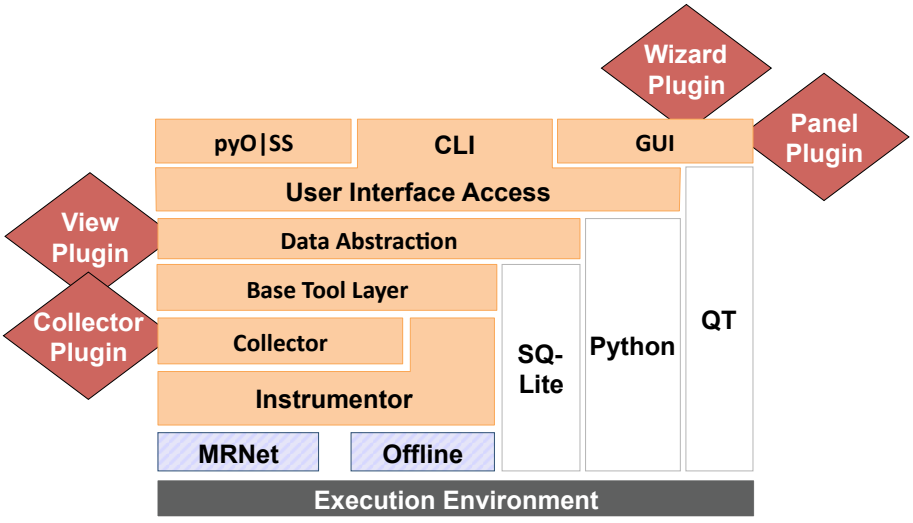

Figure 2. Open|SpeedShop framework and its open source components.

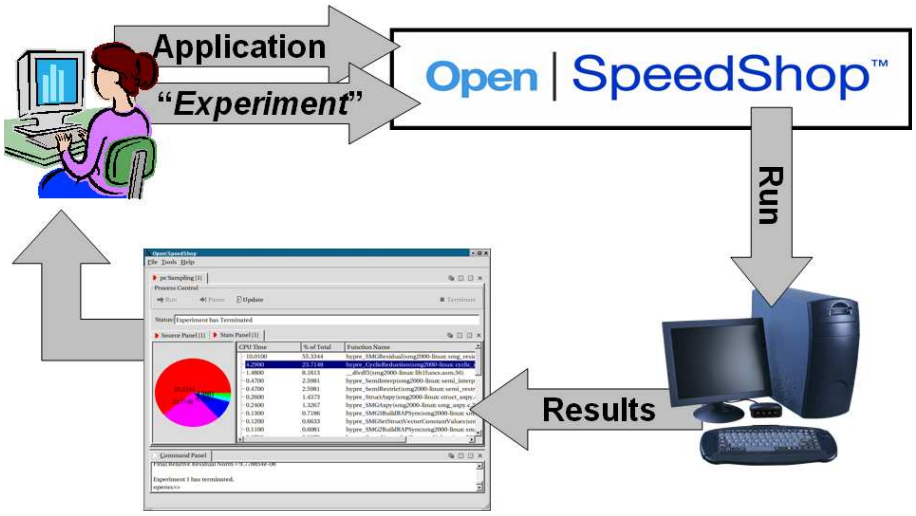

Figure 3. Typical Workflow in O|SS.

is integrated with the CLI. This component provides central access to the actual performance analysis framework. Any command from any user interface is translated into the equivalent command line command and passed to the CLI parser for processing. Similarly, any result return by a command is sent through the CLI and passed back to user interface from where the command originated.

This unique architecture not only ensures the full interoperability of all user interfaces, but also provides the additional benefit that commands in any user interface can be recorded into a script and later replayed through O|SS's batch interface. In particular, users can execute an experiment through the GUI, later recall the commands that were used by the GUI to run the selected experiment, and use them to automatically repeat the manually executed experiment. This significantly lowers the learning curve for O|SS's scripting language and enables even untrained users to create repeatable batch experiments.

Independent of which interface is used, users of O|SS generally follow the same workflow, which is depicted in Figure 3: they first select their target application, the experiment, as well as the data collection mechanism. O|SS then loads the application or attaches to it, instruments it with the necessary performance probes, and runs the application. Depending on the collection mechanism, the performance data is either stored in temporary files and converted after the run's completion, or is directly stored in an SQL database. Once the application terminates or the user polls for intermediate data in the online mode, O|SS reads the collected data from the database and then displays it to the user or performs additional analysis on it. The latter includes cross experiment comparisons, multi rank analysis, or clustering techniques. In any case, the user can use this to gather information about his/her application and then apply this information to either initiate a next set of experiments or to modify the code.

THE ARCHITECTURE OF THE O|SS FRAMEWORK

The O|SS framework, as shown in Figure 2, consists of several layers, each raising the abstraction of the performance data. Collectors are inserted into an application by the O|SS Instrumentor component using one of two instrumentation options. These

are described in more detail in the following section. Any collected data is passed through a base tool layer, which preprocesses the raw data from the collectors to a common format, to the data abstraction layer, which is responsible for storing the data in the SQLite database. When data is requested by a user through the user interface access module, the data abstraction layer is further responsible for querying the data from the database and for preparing it for display. The latter includes any necessary data aggregation, e.g., the generate sums or average of data from multiple MPI ranks.

Experiments are implemented as a set of plugins that define what data is collected (*Collector Plugin*), how it is presented to the user (*View Plugin*), and how it is displayed in the GUI (*Panel Plugin*). Additionally, an optional (*Wizard Plugin*) can extend the GUI to help guide users through the setup of that particular experiments All plugins are loaded by O|SS during startup and provide the actual tool functionality. This plugin concept can also be used to extend O|SS easily with new experiments, while inheriting all benefits from the O|SS framework.

O|SS relies on several open source infrastructures and the most important ones are shown on the right side of Figure 2. The user interface access component uses the Python parser for the processing of CLI commands. As a consequence of this design, the CLI can not only process its own commands, but any command can also be augmented by Python control flow commands, including conditionals and loop structures, which significantly increases the flexibility of the CLI language.

We use the open source database SQLite to store any performance data in the form of a relational database. We chose this particular database implementation, since it stores each database as a separate file, which allows us to provide an easy-to-use standalone tool. However, the interface within O|SS solely relies on SQL commands to access the database and hence other database implementations could be substitute SQLite without a major redesign.

All GUI components are written using the open source version of QT. This also includes any panel or wizard plugins. However, O|SS can also be built in CLI mode only, which drops the dependence on QT.

INSTRUMENTATION AND DATA COLLECTION OPTIONS

As mentioned above, O|SS currently provides two different mechanisms to introduce performance instrumentation into application binaries (Figure 4): *Offline* instrumentation at link time and *Online* or dynamic instrumentation in combination with a tree-based aggregation network based on *MRNet* [15].

Using the *Offline* variant, the data collector is loaded into the application at link time using library preload and function interception mechanisms. Any data is written to raw files, which are aggregated post-mortem and at that point translated into a single O|SS database, which can then be analyzed in the O|SS tool set.

The *Online* instrumentation uses dynamic instrumentation (provided by Dyninst [3]) to insert the data collection directly into the already loaded and running binary. Any collected data is transferred using a tree-based overlay network, implemented using MRNet, to the front-end tool and stored into an O|SS database. Additionally, it can also be displayed as it comes available allowing the user to monitor progress and to study intermediate results.

Both instrumentation options can operate on unmodified binaries and both offer the same kind of data to the user. However, they represent different tradeoffs in terms of flexibility and performance. Offline instrumentation is typically more robust

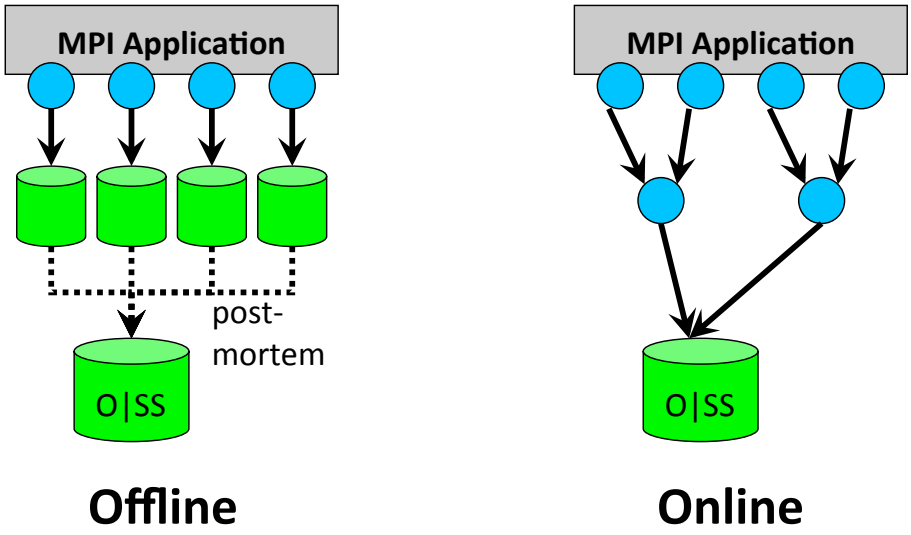

Figure 4. O|SS instrumentation options: Offline vs. MRNet.

and easier to deploy, in particular on new platforms. In most cases it also offers slightly lower overhead. However, it is restricted to end-to-end measurements, i.e., it must be enabled during the entire run of an application and results are not available before the application has terminated. The dynamic instrumentation in the MRNet-based approach, on the other hand, enables O|SS to attach performance experiments to already running applications and provides the ability to query intermediate results during the application's runtime. Additionally, this online approach eliminates the need to created intermediate raw data files. However, it comes at the cost of additional installation complexity and overhead.

Users can select the instrumentation technique based on their preferences and environment. Typically, it is recommended to use the simpler offline instrumentor when possible, i.e., if it is not necessary to attach to a running application, due to its lower overhead and higher robustness.

SCALABILITY AND OVERHEAD

Performance tools need to be designed to exhibit minimal overhead, in order to not perturb the execution of the application to a point that it significantly alters the application's characteristics. Otherwise, the results obtained from the tool do not reflect the real performance of the application and hence can lead to wrong conclusions.

To study the overhead caused by O|SS we apply it to a set of runs with SMG2000, a Semicoarsening Multigrid Solver based on the hypre library [4], taken from the ASCI Purple benchmark suite [7]. We use an input size $N = 90^3$ for all experiments and we execute them on Hyperion [8], a 1152 node Linux cluster. Each node is equipped with dual socket Intel Xeon Quad-core processors and 8GB of main memory, interconnected with Quad data rate Infiniband.

We investigate the performance of both sampling and tracing experiments and Figure I shows the results of four different experiments comparing the execution of the code with O|SS to a baseline without instrumentation. Sampling experiments are expected to produce a uniform overhead across the whole application. For O|SS we see an only negligible overhead of 0.3%-0.6% for the PC sampling and the hardware

Tasks	Nodes	PCsamp	HWC	IO	IOT
64	8	0.3%	0.2%	0.0%	1.5%
128	16	0.3%	0.3%	0.0%	0.0%
256	32	0.4%	0.4%	0.0%	0.3%
512	64	0.5%	0.6%	0.1%	2.4%
1024	128	0.4%	0.4%	-	-

TABLE I. OVERHEADS WHEN EXECUTING O|SS ON SMG2000 USING A VARIETY OF EXPERIMENTS.

counter (HWC) experiments. Further, our tool framework scales well showing only a slight increase in overhead when scaling the application run from 64 to 1024 tasks.

The performance of tracing experiments, on the other hand, depends heavily on the activity within the application that is being traced. Typically, tracing experiments capture large amounts of data and are expected to create significant overhead. The results in Figure I show the performance of the I/O tracer with and without stack tracer enabled. SMG2000 does only minimal I/O and hence the numbers reflect the overhead required by O|SS to setup the experiment and prepare the trace information, but not the actual recording of trace data. Again, we see a very low overhead and a good scalability in all cases, but using stacktraces can in some cases created a slightly higher overhead of 2.5%.

RELATED WORK

Typically, operating systems already provide a set of basic performance tools. This often includes *prof* or *gprof*, a basic profiler, and utilities like *time* to measure execution time or *strace* to track all system calls.

Besides them, a large body of research exists covering a wide range of performance tools. Examples are HPCView from the HPCToolkit [10], which provides a graphical user interface to gather a detailed performance profile of an application on a per function basis; the PAPI [13] utilities or the HPM Tool Kit [5] to gather hardware counter statistics across an application's execution; the Visual Profiler (vProf) [6]; and DynTG [16], which allows users to dynamically insert performance calipers into a running application.

Further, several tools have been developed specifically for the analysis of parallel applications. This includes both profiling tools, like mpiP [17], and tracing and trace visualization tools, like Vampir [14], Vampir Next Generation (VNG) [2], or Jumpshot [19]. SCALASCA [18] and its predecessor Kojak [12] work on both profiles and traces and offer an automated analysis of traces to identify communication bottlenecks. Systems like TAU/ParaProf [1], which provides a whole suite of performance tools and visualizers, and Paradyn [11], which focuses on the use of dynamic instrumentation, can be used both on sequential and parallel codes.

CONCLUSIONS AND FUTURE WORK

O|SS is an open-source performance analysis tool set for Linux clusters. It provides a range of experiments for sampling and tracing of performance data for sequential, MPI parallel, and multithreaded codes. Users can access the tool through

three separate, yet fully interoperable interfaces. O|SS can be applied to existing binaries without requiring any changes to source code or recompilation and can hence be applied easily to any application. Using a case study of a multigrid solver, we have shown that the overheads caused by the tool are negligible and that the framework supporting O|SS is scalable. Future work includes porting O|SS to new platforms, including machines like BG/L, BG/P, and Cray's XT line, as well as further scalability enhancements. This will develop O|SS into a true cross-platform performance analysis tool set that can be applied by end users at any step of their code development efforts, be it on commodity clusters, small test configurations, or production environments on high-end systems.

ACKNOWLEDGMENTS

Part of this work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. The project is currently funded by DOE/NNSA through ASC/CCE and by DOE/OASCR through the Peta-Tools program.

REFERENCES

1. R. Bell, A. Malony, and S. Shende. ParaProf: A Portable, Extensible, and Scalable Tool for Parallel Performance Profile Analysis. In *Proceedings of the International Conference on Parallel and Distributed Computing (Euro-Par 2003)*, pages 17–26, Aug. 2003.
2. H. Brunst, D. Kranzlmüller, and W. Nagel. Tools for Scalable Parallel Program Analysis - Vampir NG and DeWiz. *The International Series in Engineering and Computer Science, Distributed and Parallel Systems*, 777:92–102, 2005.
3. B. Buck and J. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, 2000.
4. R. Falgout and U. Yang. hypre: a Library of High Performance Preconditioners. In *Proceedings of the International Conference on Computational Science (ICCS), Part III, LNCS vol. 2331*, pages 632–641, Apr. 2002.
5. IBM-Alphaworks. HPM Tool Kit. http://www.alphaworks.ibm.com/tech/hpmtoolkit, Nov. 2001.
6. C. Jannsen. The Visual Profiler. `http://aros.ca.sandia.gov/ cljanss/`, Mar. 2002.
7. Lawrence Livermore National Laboratory. The ASCI purple benchmark codes. http://www.llnl.gov/asci/purple/benchmarks/limited/code_list.html, Oct. 2002.
8. Lawrence Livermore National Laboratory. Lawrence Livermore teams with computing industry leaders to develop an advanced technology cluster testbed. Press Release, available at https://publicaffairs.llnl.gov/news/news_releases/2008/NR-08-11-04.html, Nov. 2008.
9. A. D. Malony and W. E. Nagel. The open trace format (OTF) and open tracing for HPC. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 24, New York, NY, USA, 2006. ACM.
10. J. Mellor-Crummey, R. Fowler, and G. Marin. HPCView: A tool for top-down analysis of node performance. *The Journal of Supercomputing*, 23:81–101, 2002.
11. B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, R. Irvin, K. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn Parallel Performance Measurement Tool. *IEEE Computer*, 28(11):37–46, Nov. 1995.
12. B. Mohr and F. Wolf. KOJAK - A Tool Set for Automatic Performance Analysis of Parallel Programs. In *Proceedings of the International Conference on Parallel and Distributed Computing (Euro-Par 2003)*, pages 1301–1304, Aug. 2003.
13. P. J. Mucci, S. Browne, C. Deane, and G. Ho. PAPI: A portable interface to hardware performance counters. In *Proc. Department of Defense HPCMP User Group Conference*, June 1999.
14. W. E. Nagel, A. Arnold, M. Weber, H. C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12(1):69–80, 1996.
15. P. Roth, D. Arnold, and B. Miller. MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools. In *Proceedings of IEEE/ACM Supercomputing '03*, Nov. 2003.

16. M. Schulz, J. May, and J. Gyllenhaal. DynTG: A Tool for Interactive, Dynamic Instrumentation. In *Proceedings of the 5th International Conference in Computational Science (ICCS), Part II, LNCS vol. 3515*, pages 140–148, May 2005.
17. J. Vetter and C. Chambreau. mpiP: Lightweight, Scalable MPI Profiling. http://www.llnl.gov/CASC/mpip/, Apr. 2005.
18. F. Wolf, B. Wylie, E. Abraham, D. Becker, W. Frings, K. Fuerlinger, M. Geimer, M.-A. Hermanns, B. Mohr, S. Moore, and Z. Szebenyi. Usage of the SCALASCA Toolset for Scalable Performance Analysis of Large-Scale Parallel Applications. In *Proceedings of the 2nd HLRS Parallel Tools Workshop*, Stuttgart, Germany, july 2008.
19. O. Zaki, E. Lusk, W. Gropp, and D. Swider. Toward Scalable Performance Visualization with Jumpshot. *International Journal of High Performance Computing Applications*, 13(3):277–288, 1999.